Bemerkungen:

- Typ-Parameter dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden !!!
- Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

  `A<S,T> extends B<T>` ist erlaubt :-)

  `A<S> extends B<S,T>` ist verboten :-)

- `Poly()` ist eine Unterklasse von `Object`; aber `List<Poly>` ist keine Unterklasse von `List<Object>` !!!

Bemerkungen (Forts.):

- Für einen Typ-Parameter T kann man auch eine Oberklasse oder ein Interface angeben, das T auf jeden Fall erfüllen soll ...

```
public interface Executable {
   void execute ();
}
public class ExecutableList<E extends Executable> {
   E element;
   ExecutableList<E> next;
   void executeAll () {
        element.execute ();
        if (next == null) return;
        else next.executeAll ();
}  }
```

Bemerkungen (Forts.):

- Beachten Sie, dass hier ebenfalls das Schlüsselwort extends benutzt wird!
- Auch gelten hier weitere Beschränkungen, wie eine parametrisierte Klasse eine Oberklasse sein kann :-(
- Auch Interfaces können parametrisiert werden.
- Insbesondere kann Comparable parametrisiert werden – und zwar mit der Klasse, mit deren Objekten man vergleichen möchte ...

```
public class Test implements Comparable<Test> {
   public int compareTo (Test x) { return 0; }
}
```

14.3 Wrapper-Klassen

... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!

  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl :-)

14.3 Wrapper-Klassen

... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!

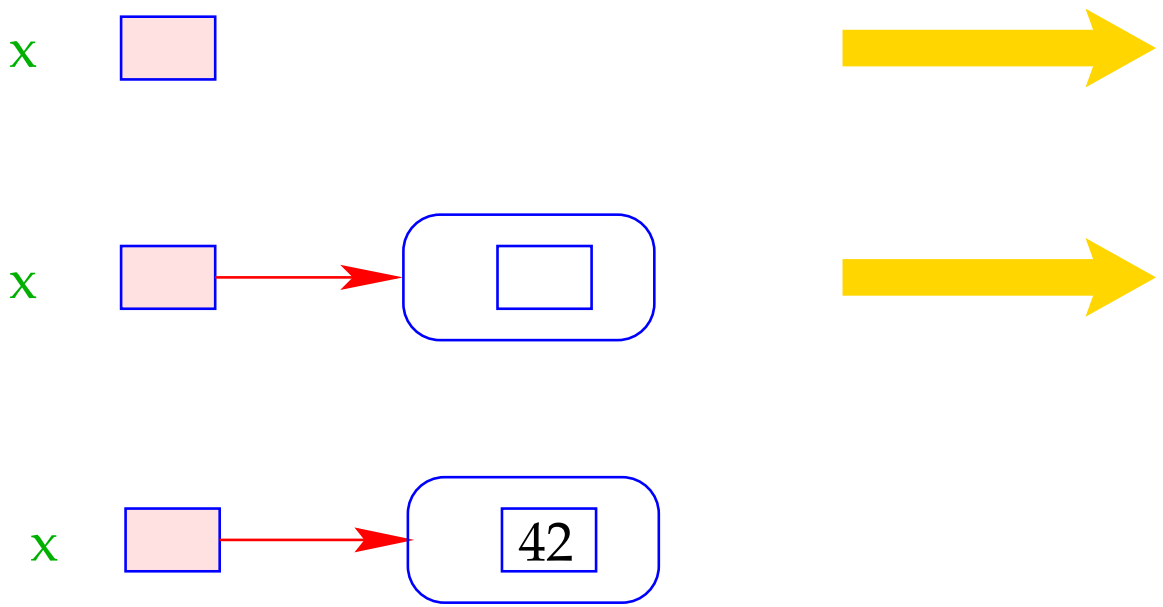
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl :-)

Ausweg:

- Wickle die Werte eines Basis-Typs in ein Objekt ein!

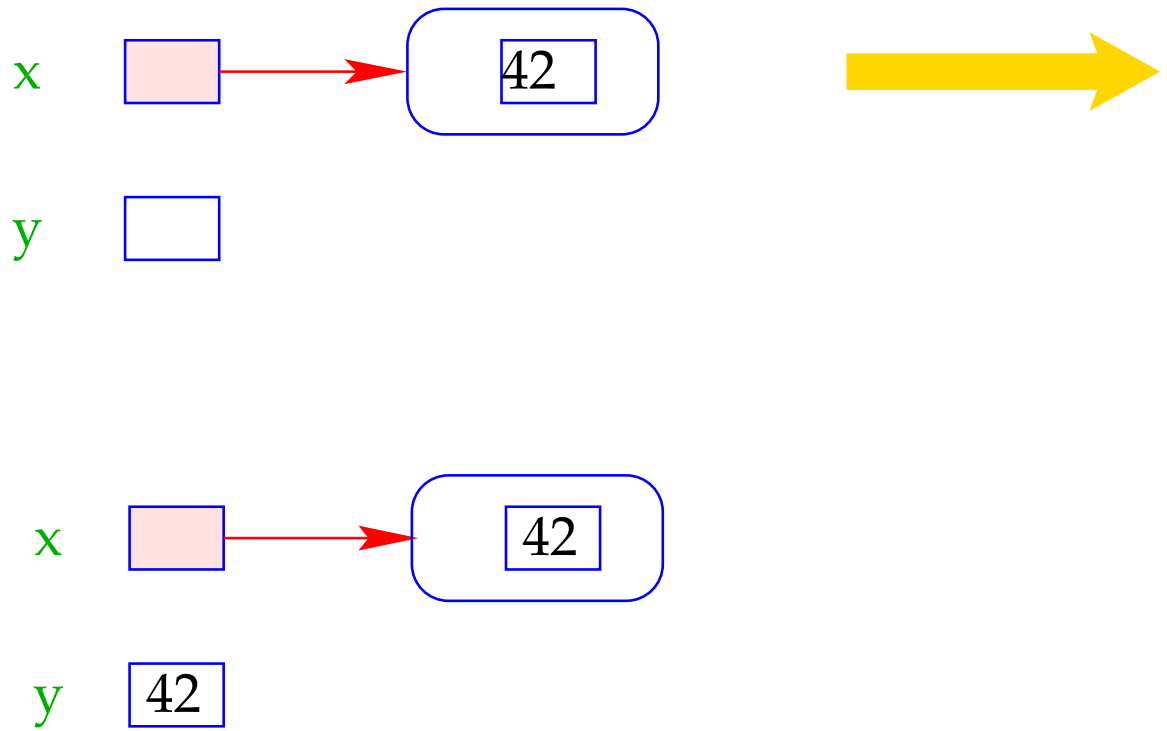
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

Die Zuweisung `Integer x = new Integer(42);` bewirkt:

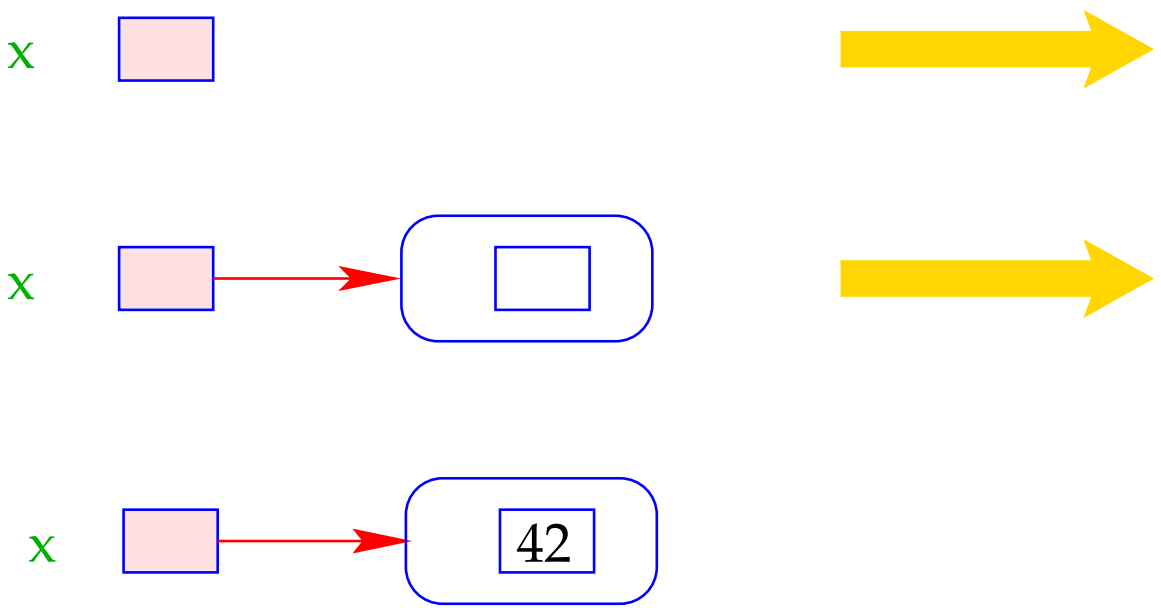


Eingewickelte Werte können auch wieder ausgewickelt werden.

Seit Java 1.5 erfolgt bei einer Zuweisung `int y = x;` eine automatische Konvertierung:



Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

Zum Beispiel:

- `public static int MIN_VALUE = -2147483648;` liefert den kleinsten `int`-Wert;
- `public static int MAX_VALUE` = 2147483647; liefert den größten `int`-Wert;
- `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

  Andernfalls wird eine ↑exception geworfen :-)

Bemerkungen:

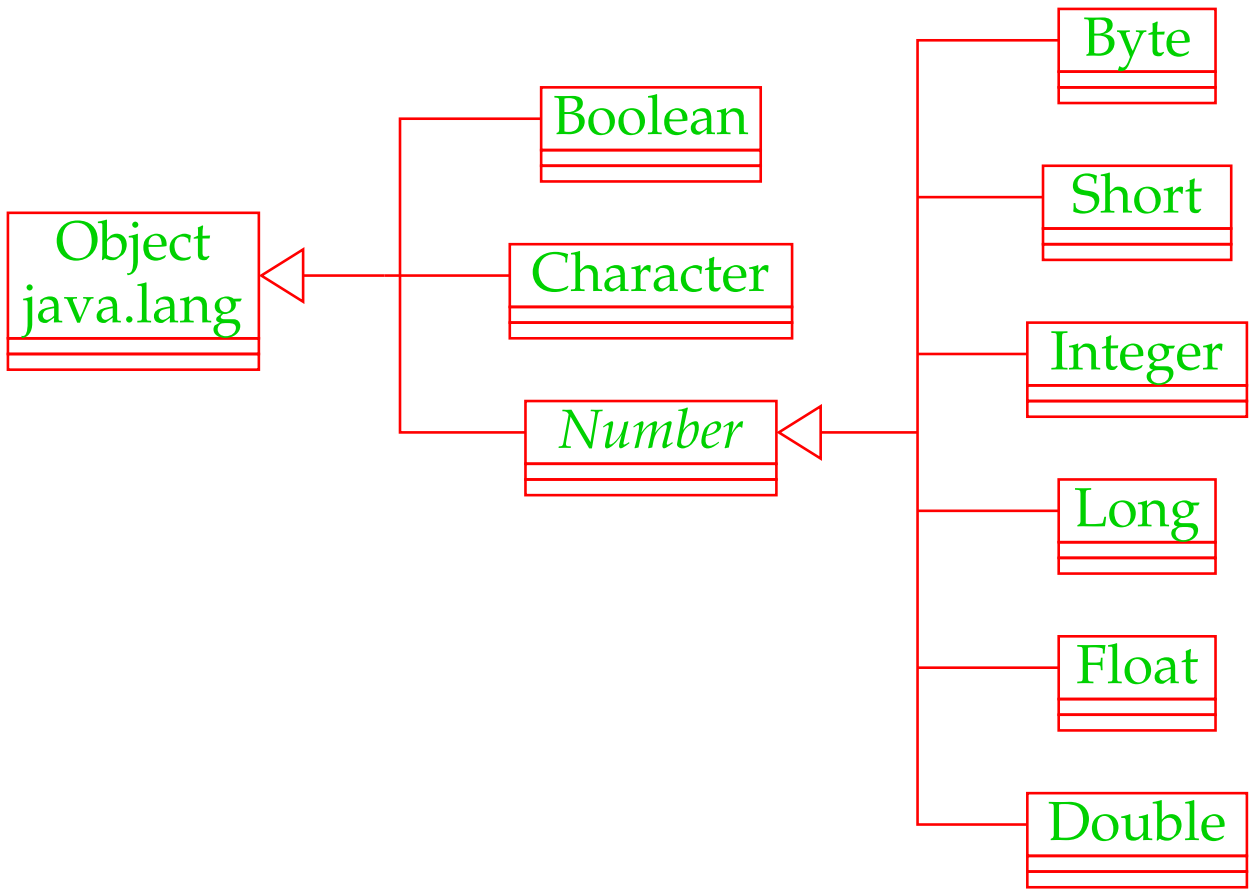
- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

Bemerkungen:

- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen ...

Wrapper-Klassen:



- Sämtliche Wrapper-Klassen für Typen type (außer `char`) verfügen über
  - Konstruktoren aus Basiswerten bzw. `String`-Objekten;
  - eine statische Methode `type parseType(String s);`
  - eine Methode `boolean equals(Object obj)` (auch `Character`).
- Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln ...
- Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- Diese Klasse ist ↑abstrakt d.h. man kann keine `Number`-Objekte anlegen.

Spezialitäten:

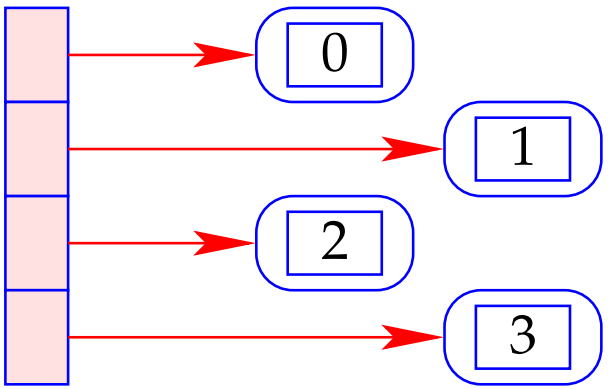
- `Double` und `Float` enthalten zusätzlich die Konstanten

`NEGATIVE_INFINITY`	=	-1.0/0
`POSITIVE_INFINITY`	=	+1.0/0
`NaN`	=	0.0/0

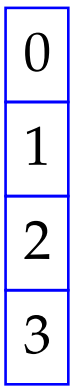
- Zusätzlich gibt es die Tests
  - `public static boolean isInfinite(double v);`
    `public static boolean isNaN(double v);`
    (analog für `float`)
  - `public boolean isInfinite();`
    `public boolean isNaN();`

  mittels derer man auf (Un)Endlichkeit der Werte testen kann.

Vergleich `Integer` mit `int`:



`Integer []`



`int []`

\+ `Integers` können in polymorphen Datenstrukturen hausen.

– Sie benötigen mehr als doppelt so viel Platz.

– Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten $\Longrightarrow$ schlechteres Cache-Verhalten.

15 Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche :-| Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.

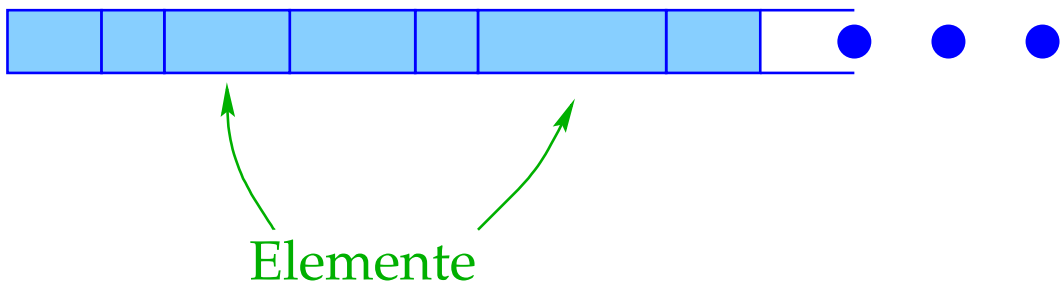
15 Ein- und Ausgabe

- Ein- und Ausgabe ist nicht Bestandteil von Java.
- Stattdessen werden (äußerst umfangreiche :-| Bibliotheken von nützlichen Funktionen zur Verfügung gestellt.

Vorteil:

- Weitere Funktionalität, neue IO-Medien können bereit gestellt werden, ohne gleich die Sprache ändern zu müssen.
- Programme, die nur einen winzigen Ausschnitt der Möglichkeiten nutzen, sollen nicht mit einem komplexen Laufzeit-System belastet werden.

Vorstellung:



- Sowohl Ein- wie Ausgabe vom Terminal oder aus einer Datei wird als Strom aufgefasst.
- Ein Strom (Stream) ist eine (potentiell unendliche) Folge von Elementen.
- Ein Strom wird gelesen, indem links Elemente entfernt werden.
- Ein Strom wird geschrieben, indem rechts Elemente angefügt werden.

Unterstützte Element-Typen:

- Bytes;
- Unicode-Zeichen.

Achtung:

- Alle Bytes enthalten 8 Bit :-)
- Intern stellt Java 16 Bit pro Unicode-Zeichen bereitgestellt ...
- standardmäßig benutzt Java (zum Lesen und Schreiben) den Zeichensatz Latin-1 bzw. ISO8859_1.
- Diese externen Zeichensätze benötigen (welch ein Zufall :-) ein Byte pro Zeichen.

Orientierung:

- Will man mehr oder andere Zeichen (z.B. chinesische), kann man den gesamten Unicode-Zeichensatz benutzen.
- Wieviele Bytes dann extern für einzelne Unicode-Zeichen benötigt werden, hängt von der benutzten Codierung ab ...
- Java unterstützt (in den Klassen `InputStreamReader`, `OutputStreamReader`) die UTF-8-Codierung.
- In dieser Codierung benötigen Unicode-Zeichen 1 bis 3 Bytes.

Problem 1: Wie repräsentiert man Daten, z.B. Zahlen?

- binär codiert, d.h. wie in der Intern-Darstellung

  $\Longrightarrow$ vier Byte pro `int`;

- textuell, d.h. wie in Java-Programmen als Ziffernfolge im Zehner-System (mithilfe von Latin-1-Zeichen für die Ziffern)

  $\Longrightarrow$ bis zu elf Bytes pro `int`.

	Vorteil	Nachteil
binär	platzsparend	nicht menschenlesbar
textuell	menschenlesbar	platz-aufwendig